Praktikumsbericht

Firmware Upgrade des PowerIt-Boards

Patrick Nisblé

6. April 2018

Zusammenfassung

Um das PowerIt um einige Funktionen erweitern zu können wurde in diesem Praktikum eine Entwicklungsumgebung zusammengestellt (basierend auf Open-Source Projekten).

Die alte Firmware ist auf eine neue Version portiert worden. Und die so erstellte Firmware (basierend auf ChibiOS) kann die Bisherige, im BrainScaleS System eingesetzte, ersetzen.

Inhaltsverzeichnis

1. Ziel

Der erste Teil des Praktikums, bestand im Wesentlichen aus der Suche nach geeigneten Tools, die die Arbeit an der Firmware und das Programmieren des Mikrocontrollers ermöglichen. Dabei steht im Vordergrund, dass die Entwicklungsumgebung auf Open-Source Projekten basieren soll.

Im zweiten Teil soll die bisherige Firmware, durch eine Neuere ersetzt werden. Dafür soll die neue Entwicklungsumgebung genutzt werden.

2. Einleitung

Die Forschung der Gruppe Electronic Vision(s) umfasst alle Komponenten sowie Softwarelösungen rund um das Wafersystem BrainScaleS (Abb. 1).

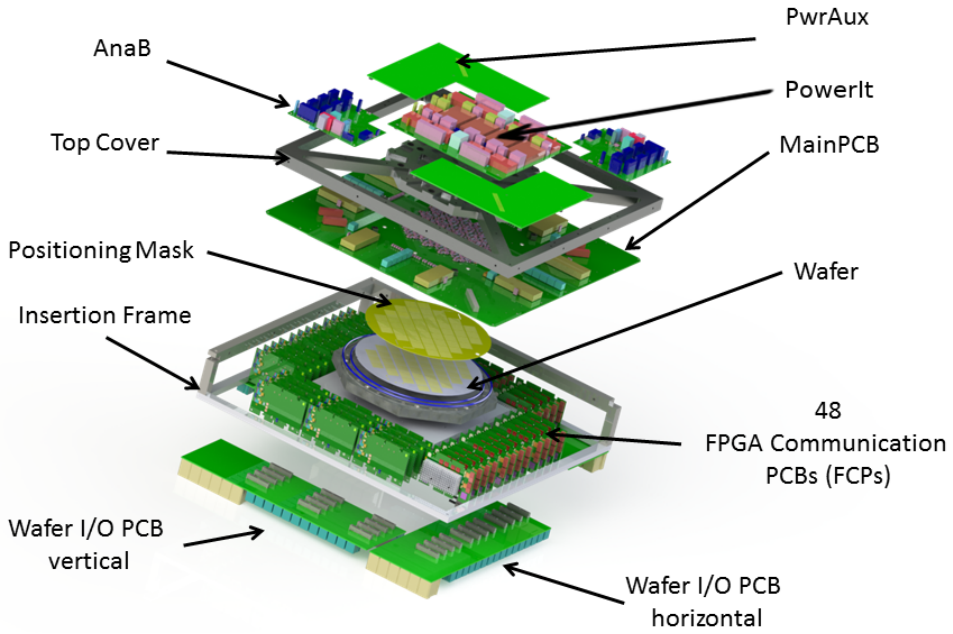


Abbildung 1: Das BrainScaleS System, zu sehen sind die Bausteine eines Wafersystems, sowie Kommunikations- und Stromversorgungskomponenten [1]. Das PowerIt liefert die Stromversorgung für HICANNs und FPGAs.

Versorgt werden diese Komponenten mit 1.8V (HICANNs) und 9.6V (FPGAs[1]). Die gesamte Leistungsaufnahme dieser Komponenten kann dabei etwa 2kW (pro Wafer-System) erreichen. Eingesetzt wird hierfür ein spezialisiertes Board, das PowerIt.

Ein Wafersystem beinhaltet zudem einen Raspberry Pi, der das System überwacht, unter anderem auch das PowerIt.

[1] Field Programmable Gate Array

3. Das PowerIt

Das Board und dessen Komponenten sind der Gegenstand dieses Praktikums. Der zentrale Kern des PowerIt Boards ist ein Mikrocontroller [2] der Firma STMicroelectronics [3].

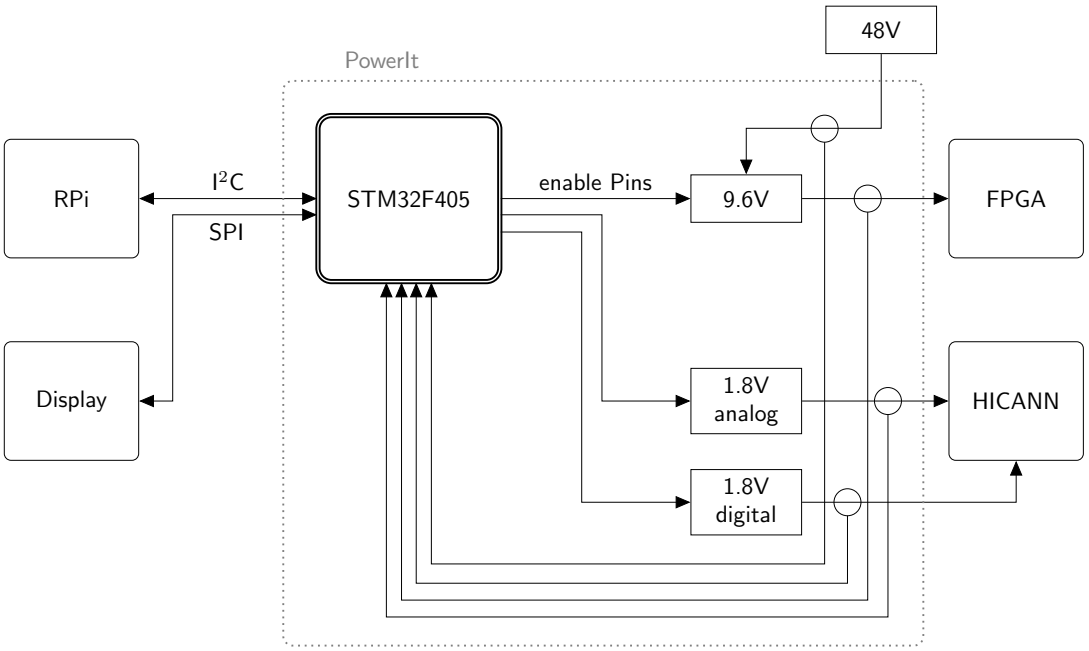


Abbildung 2: Verbindungen des STM32-Chips, zu sehen sind die durch enable-Pins kontrollierten 1.8V und 9.6V Spannungsregler, die Messstellen (für Strom und Spannung) des Boards, sowie das Monitoring durch einen Raspberry Pi und die Debugging-Schnittstelle (über USART [3]).

Betrachtet man die schematische Darstellung in Abb. 2, so sieht man die Platzierung des PowerIt im System und eine vereinfachte Darstellung der Aufgaben des Chips:

- Kontrolle der stromversorgenden Komponenten (über enable-Pins)
- Messen der Spannung und des Stroms auf den Leitungen (verbunden mit einem internen ADC[4]),
- Kommunikation mit dem Monitoring-System (über I^2C-Bus)
- und der Debugging-Schnittstelle (über USART Bus)

[2] **STM32 F405 RGT6[2]:** ARM Cortex-M4 32b MCU+FPU, 168 MHz Clock, 1MB Flash, 192kB SRAM, 3 12-Bit ADCs, GPIO with Interrupts[2]
[3] Universal Synchronous/Asynchronous Receiver/Transmitter
[4] Analog to Digital Converter

4. Evaluation möglicher Buildchains/-Umgebungen

Vorab definieren wir einige Kriterien, die die Firmware erfüllen soll:

- RTOS[5] mit Threading System und Prioritäten
- nutzt den HIL[6] um kritische Zustände zu priorisieren
- implementiert ein HAL[7]

Die gefundenen Softwarelösungen sind im Folgenden untersucht.

4.1 PlatformIO

Ein erster Ansatz mit dem Versprechen einer Komplettlösung kann in diesem Fall das Projekt PlatformIO [4] bieten:

build-tool: scons

compiler: gcc-arm (v6)

programming: st-link [5]

PlatformIO unterstützt die STM32 Platform von sich aus und man benötigt nur wenig Konfiguration um einfachen Code auf einem Chip laufen zu lassen. Dazu unterscheidet man mögliche Frameworks, die für die STM32-Chips zur Verfügung stehen:

CMSIS: "ARM Cortex Microcontroller Software Interface Standard" [6],

- low-level Abstraktion
- entwickeln einfacher Systeme

✓ HAL

✓ HIL

mbed: "for complex projects, built on low-level CMSIS API" [7],

- großer Umfang
- bessere Abstraktion als CMSIS
- komplexe Bibliotheken (z.B USB und Netzwerk)

✓ HAL

✓ HIL

(✓) RTOS

[5] Real Time Operating System
[6] Hardware Interrupt Layer: Reagieren auf Hardware Interrupt Pins
[7] Hardware Abstraction Layer: Softwareseitige Vereinheitlichung der Hardware

STM32Cube: “light-weight, optimized, [...] APIs” [8],

- portabel zwischen STM32-Chips
- optimiert für STM32-Chips
- direkte Lösung von STMicroelectronics

✓ HAL

✓ HIL

Es bietet sich hier keines der möglichen Frameworks an. Alle Frameworks die kein RTOS implementieren, das im Rahmen von PlatformIO nutzbar ist, entfallen als Möglichkeit. Dadurch fallen CMSIS und STM32Cube weg. Das mbed-Framework bietet als Einziges ein RTOS, mbed OS [9], ist aber in unserer Anwendung der Alternative ChibiOS unterlegen.

4.2 Der ChibiOS Weg (make + gcc-arm + st-link)

Das zuvor verwendete RTOS ChibiOS scheint eine mögliche Lösung zu sein. Es unterstützt die folgende Softwarekombination:

build-tool: waf[10] (make)

compiler: gcc-arm (v6)

flashing: st-link

Das mit ChibiOS ausgelieferte Makefile ist eine gute Grundlage auf der aufgebaut werden kann und es entspricht den oben definierten Kriterien. Außerdem existiert alter Code der auf ChibiOS basiert und somit portierbar ist.

5. “One RTOS to rule them all”

ChibiOS [11] ist eine Firmware für Embedded Systeme[8]. Es arbeitet dabei sehr gut auf den STM32 Chips und beinhaltet einige Demos die auf diesen Chips laufen.

Das ChibiOS Projekt besteht aus 4 Teilen:

ChibiOS/HAL ist der HAL von ChibiOS und Grundgerüst für die restlichen Teile

ChibiOS/EX ist das Peripherie-Pendant zum HAL

ChibiOS/nil ist der minimal Kernel und die Alternative zum RT-Kernel, wenn einige Teilsysteme nicht benötigt werden

ChibiOS/RT ist der RTOS Kernel, der die High-Level Implementationen umfasst

Genutzt werden in diesem Projekt nur ChibiOS/RT und ChibiOS/HAL. Betrachtet man ChibiOS genauer kann man es in fünf Ebenen zerlegen (Abb. 3), die unterschiedlich[9] sind.

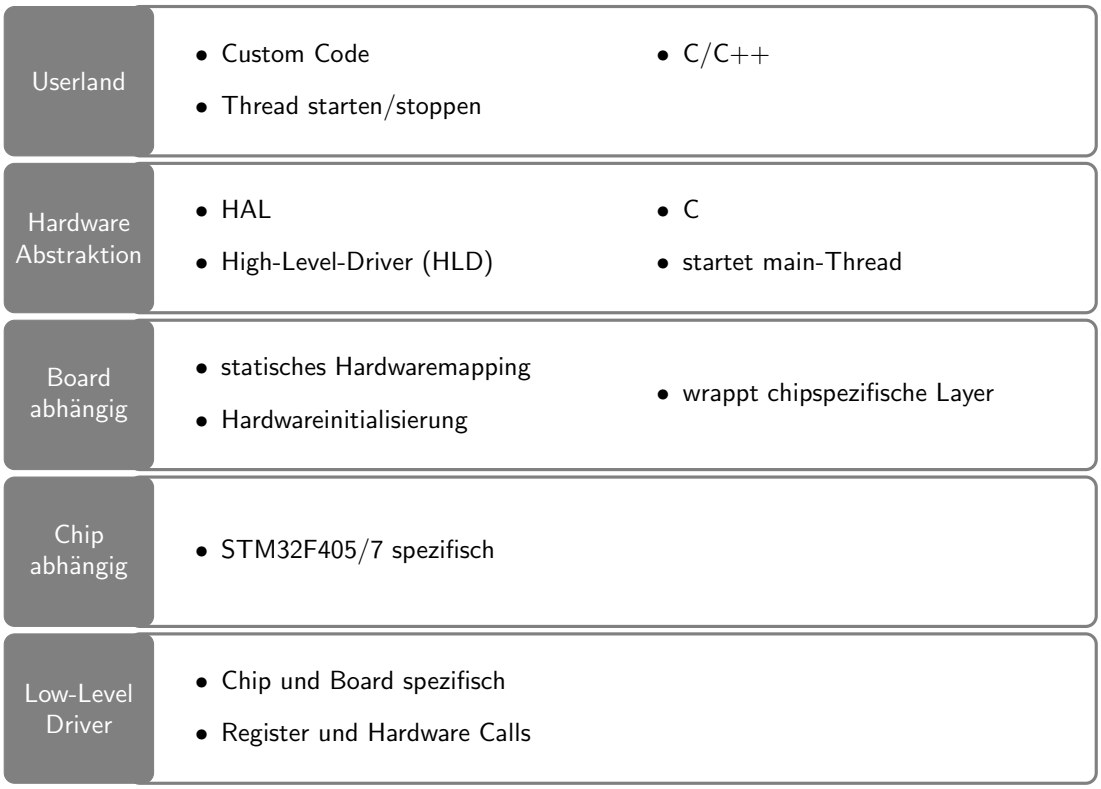


Abbildung 3: Abstraktionsebenenen von ChibiOS/RT und HAL

[8] `https://en.wikipedia.org/wiki/Embedded_system`

[9] z.B chipspezifische Hardware Calls und Register Schriebe

6. Test Phase

Um in ChibiOS einzusteigen, werden nun Testaufbauten genutzt um die Funktionalität von ChibiOS zu testen und die der alten Firmware zu reproduzieren.

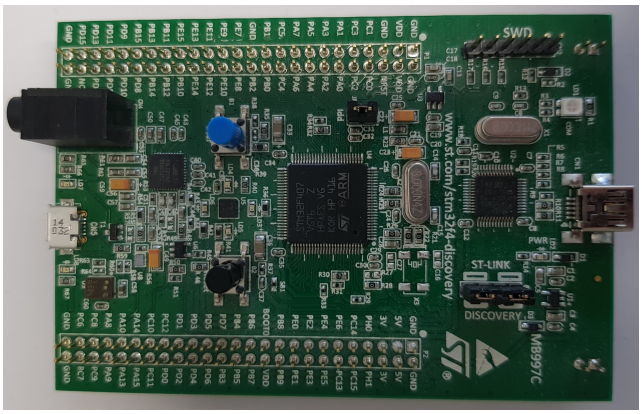


Abbildung 4: STM32F407-Discovery Board

Der erste Testaufbau (Abb. 4) besteht aus einem STM32F407-Discovery Board und einem Raspberry Pi (für I^2C und USART). Gründe dafür:

- dem im PowerIt verbauten F405-Chip sehr ähnlich
- einfacher Flashvorgang
- simples Hardware Setup

Der endgültige Testaufbau (Abb. 5) besteht aus einem PowerIt (1. Generation[10]), einem Raspberry Pi und dem STM32F407-Discovery Board. Dabei dient der Raspberry Pi als Monitoring und Debugging Gegenstück (Verbindungen wie in Abb. 6). Das Discovery Board wird genutzt zum Flashen[11] (Verbindungen wie in Abb. 7).

[10] Die Version 2 des PowerIt ist die momentan in BrainScaleS eingesetzte und mit der Vorgängerversion abwärtskompatibel.

[11] USB -> Discovery -> Serial Wire Debug (SWD) -> PowerIt

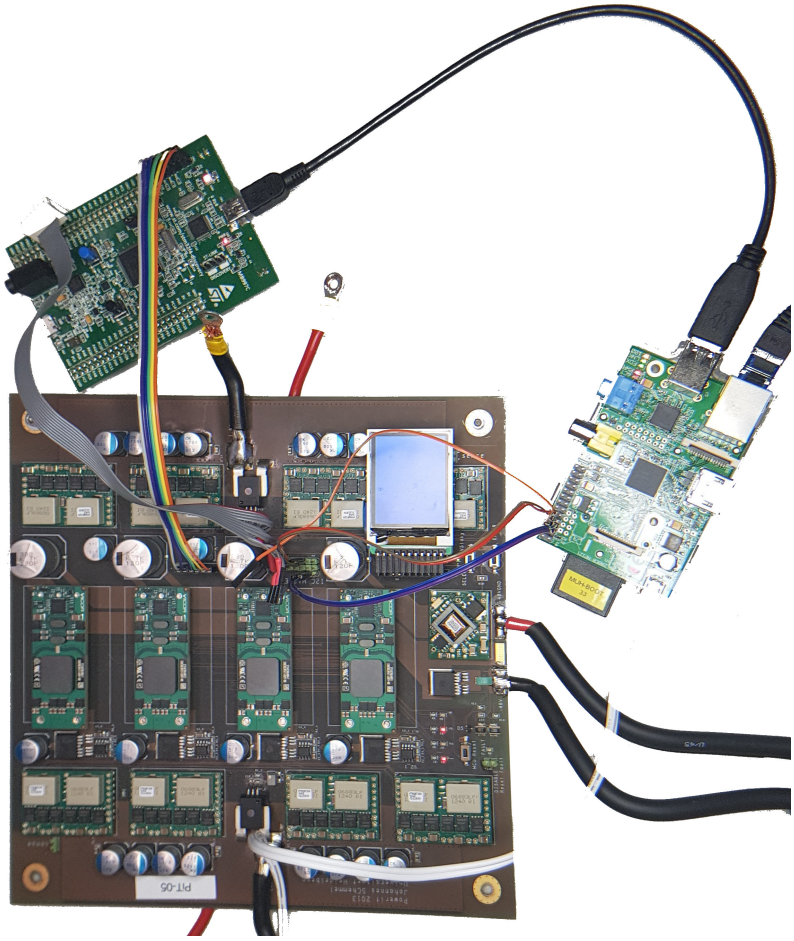


Abbildung 5: Testaufbau aus PowerIt, Raspberry Pi (rechts im Bild) und STM32F407-Discovery (oben im Bild), das PowerIt wird von einer externen Spannungsquelle mit 48V versorgt (rechts außerhalb des Bildes).

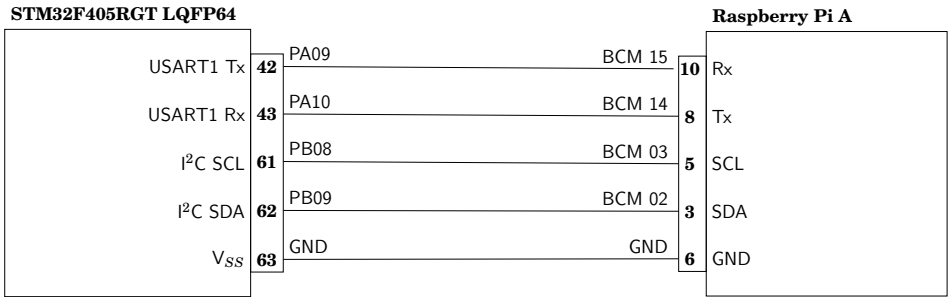


Abbildung 6: Pin Verbindungen im zweiten Testaufbau (seit sw-stm32@6f1515f0); Innen: logische Zuweisung, Mittig: Pins am Chip/Header, An den Leiterbahnen: Pinbeschriftungen; I^2C Bahnen intern mit Pullup-Widerständen verbunden.

6.1 ChibiOS 17.6.4

Als erste Einführung in ChibiOS[12]und Mikrocontroller Programmierung werden zuerst einzelne Komponenten des RTOS auf dem Discovery Board ausprobiert (Abb. 4). Teile des Systems, wie Threading, Serieller Bus und die Konsole sind dabei implementiert.

6.2 ChibiOS 2.5.2

Um die bisherige Firmware zu reproduzieren wird der alte Code benutzt. Dieser enthält die auf ChibiOS angewendeten Patches, sowie die gesamte Funktionalität der aktuell in BrainScaleS genutzten Firmware.

Nach erfolgreichem Patchen der Firmwarekomponenten und ausfindig machen der genutzten Cross-Compiler Version (GNU Tools ARM Embedded 4.7 2013q1), kann nun die Firmware erfolgreich kompiliert werden.

6.3 ChibiOS 18.2.1

Die neuere Version von ChibiOS bietet, im Gegensatz zu Version 2.5, einige Vorteile, die dazu führen Diese zu nutzen und die alte Funktionalität zu portieren:

- neue Features für dieses Projekt:
  - vervollständigter C++-Wrapper
  - ADC Messungen schneller per PWM steuerbar
  - schnellerer ADC-LLD
  - Interrupt Handling Funktionalität im Userland änderbar
  - UART → USART
  - Sicherheit dynamischer Threads
  - erweitertes Thread Schlafen und Aufwachen
  - starten statischer Threads schneller
  - inklusive FatFS Bibliothek
- Vereinheitlichung der Benennung vieler Variablen, Makros und Funktionen
- aktualisierte Dokumentation [12]

Auf der Basis dieser Gründe wird das alte Userland auf die neue ChibiOS Version portiert.

[12] sw-stm32@be67614f

7. Nutzung des sw-stm32 Repos

7.1 Abhängigkeiten

Um die Software zu nutzen, müssen folgende Module[13] geladen werden:

```
$ module load waf
$ module load gcc-arm/6
$ module load stlink
```

für das `stlink` Modul werden zusätzliche `udev` Rechte benötigt:

```
# /etc/udev/rules.d/8x-stm32.rules
SUBSYSTEMS       == "usb",
ATTRS{idVendor}  == "0485",
ATTRS{idProduct} == "3748",
MODE             := "0666"
```

7.2 Klonen

Visionary-Waf ermöglicht es die nötigen Software Repos `sw-stm32` und `lib-chibios` zu klonen:

```
$ cd <path/to/local/project>
$ waf setup --project=sw-stm32
$ waf configure --board=<project_name>
```

ab jetzt kann man die Repos behandeln wie jedes andere Projekt, das im Workflow der Arbeitsgruppe über Gerrit und Gitviz verteilt wird.

7.3 Build

Um die Firmware zu kompilieren:

```
$ cd <path/to/local/project>
$ waf [clean] build
```

und die bei erfolgreichem Kompilieren erzeugte Binary findet man danach hier:

```
<path/to/local/project>/build/sw-stm32/<project_name>.bin
```

[13] aus wang/env

7.4 Flashen

Der Flashvorgang benötigt ein Modul, z.B. das STM32F407-Discovery Board, sowie Pin Verbindungen wie in Abb. 7

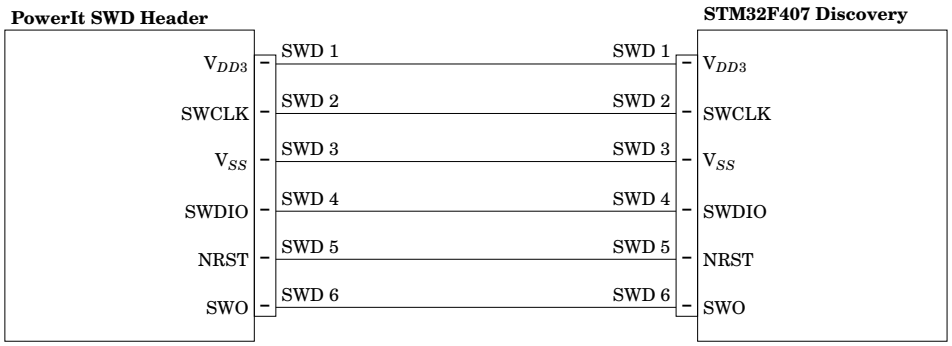


Abbildung 7: Pin Verbindungen der SWD Header, zum Flashen

Das Flashen erfolgt dann mit:

```
$ st-flash [--reset] write <path/to/binary>/<project_name>.bin 0x08000000
```

7.5 Continuous Integration

Die Software und Firmware Komponenten sind in den aktuellen Buildflow der Arbeitsgruppe eingeflossen. Dabei ist ein Jenkins CI Job (`bld_gerrit_sw-stm32`) für das PowerIt Projekt eingerichtet worden. Dieser Job reagiert auf jegliche Änderungen in der Code Review.

Momentan besteht die Verifikation durch Jenkins im Klonen und Kompilieren des in `sw-stm32` und `lib-chibios` befindlichen Source Codes.

8. PowerIt Firmware Submodule

Im Folgenden beschrieben sind die portierten Submodule des bisherigen Systems[14].

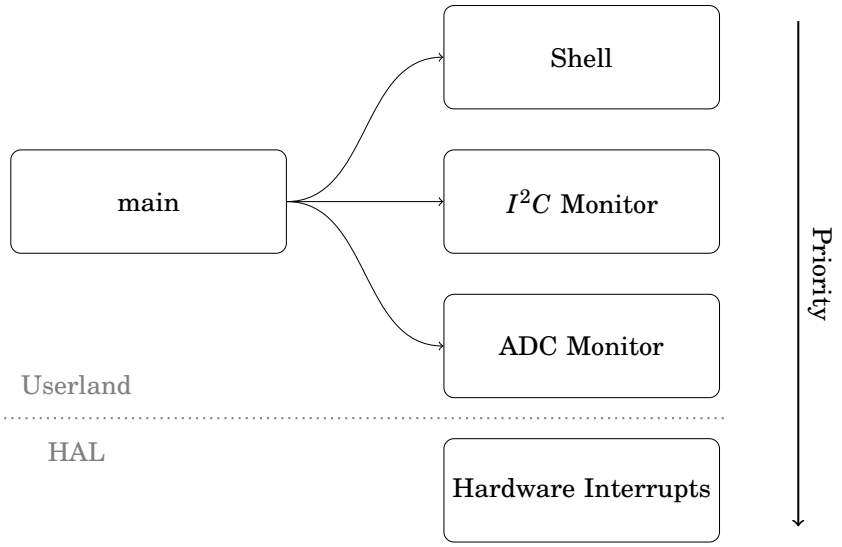


Abbildung 8: Thread/Submodul Spaltung der Firmware, mit Initialisierungshierarchie (Pfeile), sowie Prioritäten der Submodule (von oben nach unten)

Hier ist der `main`-Thread der Haupteinstiegspunkt der Firmware und initialisiert dann die HAL-Komponenten (z.B. Hardware Interrupts) und die anderen Threads.

Submodul spezifische Funktionen sind dabei in Namespaces geordnet:

`powerit::` (powerit.cpp / powerit.hpp) PowerIt spezifisch, nicht in Threads

`adcthread::` (adcthread.cpp / adcthread.hpp)

`pi2c::` (pi2cproto.hpp) I^2C-Protokoll

`chibios_rt::` (ch.hpp) ChibiOS C++ Wrapper

8.1 USART

Der USART Bus wird genutzt für Debugging Zwecke und liefert eine Konsole (läuft im dynamischen `Shell`-Thread, gestartet in `main`, s. Abb. 8). In welcher auf generelle Informationen über ChibiOS sowie das PowerIt zugegriffen werden kann. Zudem lassen sich diese Befehle erweitern und bieten eine Schnittstelle zum manuellen Ein- und Ausschalten der enable-Pins (Abb. 2).

[14] sw-stm32@dd3cc7cf

Wichtige Befehle:

- `help`: listet alle Befehle
- `info`: listet Informationen über u.a. den Firmware Build (Commit, Dirtyflag, Compiler, Compilertime) und das System (Flashgröße)
- `on|off <Bitmaske>`: schalten die in der Bitmaske selektierten enable-Pins an/aus, Bitmaske: `[1V8_digi 1V8_ana 9V6_3 9V6_2 9V6_1 9V6_0]`

8.2 PAL Konfiguration

Innerhalb der ChibiOS-Konfiguration werden den GPIO-Pins Funktionen zugewiesen. Diese können entweder an einen `Driver` gebunden, mittels des `PAL_MODE_ALTERNATE(n)` Makros (Moden s. [2] S62ff), oder strikt als IO genutzt werden.

So müssen etwa die enable-Pins der 1.8V Leitung so konfiguriert werden:

```
// hwconf.c
palSetPadMode(
    GPIOA,                          //GPIO Bank, hier A
    GPIOA_P00_VDD_1V8_DIGI,         //Pin in Bank hier 0
    PAL_MODE_OUTPUT_PUSHPULL        //GPIO Register Maske
);
```

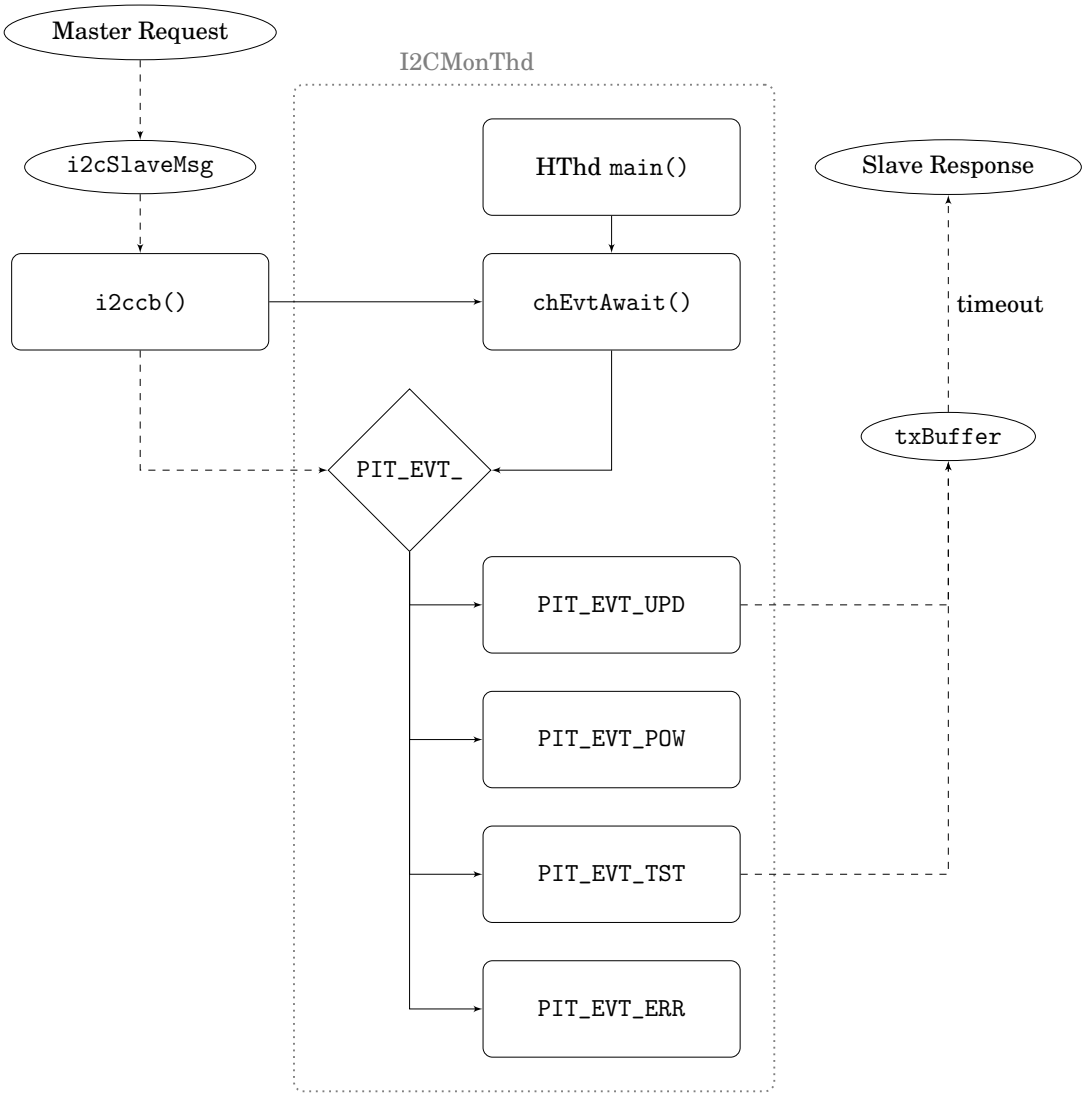
8.3 ADC Sampling

Das ADC Sampling erfolgt in `ADCMonThd`. Es werden über `ADC1` 13 Pins ausgelesen, alle mit 12 Bit Auflösung, was 4 mal wiederholt wird. Der Thread mittelt [15] diese und schreibt sie in ein `adcdata_t` Objekt. Daraus werden die in Volt und Ampere umgerechneten Werte bestimmt (`adcdata_calib_t`). Diese Daten können dann über den I^2C-Bus vom Raspberry Pi ausgelesen werden.

8.4 I^2C-Slave

Die I^2C Kommunikation stellte sich als aufwendige Implementation heraus. ChibiOS beinhaltet zwar I^2C Funktionalität, allerdings nur als Master und es fehlen low-level Komponenten, die die Slave-Funktionalität erfordert. Die Implementierung basiert auf einer, aus einem Forums Beitrag [13] erhaltene Variante des `LLD`. Diese ist für eine ältere Version von ChibiOS ausgelegt, und es sind einige Änderungen im `LLD` selbst vorzunehmen. Viele Makros und Funktionen von ChibiOS wurden umbenannt oder entfernt, andere Strukturen des RTOS, wie Threading sind verändert worden. Genutzte Funktionen im LLD können das System aufhängen, oder nicht starten lassen. Außerdem mussten einige Routinen des Forumscodes entfernt und umgeschrieben werden. Dieser ist sonst nicht mit dem genutzten I^2C-Protokoll kompatibel.

[15] nach 4 Messungen



I2CMonThd Event	pi2c::cmd_t	Antwort	Absicht
PIT_EVT_UPD	0: pi2c::cmd_update	Ja	fragt aktuelle kalibrierte ADC Werte ab
PIT_EVT_POW	1: pi2c::cmd_set_power	Nein	setzt enable-Pins
PIT_EVT_TST	2: pi2c::cmd_test	Ja	frag aktuelle ADC Werte ab
PIT_EVT_ERR	-	Nein	fehlerhafte Übertragung (CRC[16]basiert)

Abbildung 9: I^{2}C-Slave Response Prozess und zugehöriges Master-Command Protokoll [sw-stm32@8edbf9cd]

[16]Cycling Redundancy Check

Nach erfolgreicher Integration, meldete sich das PowerIt dem Raspberry Pi als I^2C Slave mit seiner Adresse (`#define slaveI2Caddress 0x10`). Um das bisherige I^2C-Protokoll nutzen zu können (s. Abb.10) wird der `i2cMonThd` genutzt. Dieser reagiert auf einen Interrupt durch die initialisierte I^2C Kommunikation, mit unterschiedlichen Funktionen, abhängig vom empfangenen Typ (`pi2c::cmd_t`).

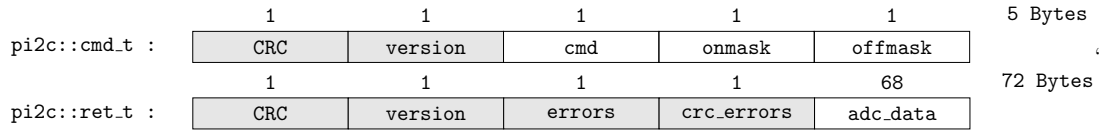


Abbildung 10: Befehl- und Antwortblöcke; Metadaten (grau) , ADC Werte: 17 Fließkommazahlen[17].

Das Protokoll (Abb. 10) basiert auf:

Master-Kommando (`pi2c::cmd_t`): vom Master gesendet, 8 Byte groß

Slave-Antwort (`pi2c::ret_t`): vom Slave zurück, 72 Byte groß

mit den folgenden Komponenten:

CRC: 7 Bit CRC der folgenden Bytes

version: Versionsnummer, Kompatibilitäts-Check

cmd: Master Kommando Typ (s. Tab. in Abb. 9)

onmask / offmask: Bitmasken für enable-Pins

errors: Identifier für Fehler beim letzten Master-Kommando

crc_errors: Identifier für CRC basierte Fehler

adc_data: momentane Werte des `ADCMonThds` + Offsets (`adcdata_calib_t`)

```
// pi2cproto.hpp
struct adcdata_calib_t {
    float 1V8_DIGI, 1V8_ANA, 1V8_IDIGI, 1V8_IANA, \
        MON_48V, MON_I48V, MON_8V_0, MON_8V_1, \
        MON_8V_2, MON_8V_3, MON_8VBUS, \
        EXT_AIN, TEMP;
    float MON_48V_OFF, MON_I48V_OFF, 1V8_DIGI_OFF, 1V8_DIGI_OFF;
};
```

[17] 13 Messungen, 4 Offsets

9. Zusammenfassung und Schlüsse

In diesem Praktikum wurde eine Softwarekombination zusammengestellt, mit der die PowerIt-Firmware weiterhin aktualisiert werden kann und diese Firmware auf einen neuen Stand gebracht.

In der Firmware enthalten sind die Komponenten, die für den bisherigen Betrieb im BrainScaleS System benötigt werden, allerdings gibt es einige Funktionen, die bis jetzt noch nicht auf dem PowerIt implementiert worden sind oder in Buildflow fehlen:

- Erweiterung des I^2C-Protokolls

  Die momentane Kommunikation basiert darauf, dass der Slaves immer die vollständige, 72 Byte lange Antwort zurück gibt. Um die I^2C Kommunikation zu erweitern und zu einem späteren Zeitpunkt nicht hunderte Byte große Pakete zu verschicken, soll die Kommunikation es erlauben einzelne Werte abzufragen.

- SPI Kommunikation mit Digitalpotentiometer

  Auf den Leitungen vom PowerIt zum HICANN findet ein Spannungsabfall statt, für den bisher keine Kompensation in das System integriert worden ist. Das PowerIt (v0.2) besitzt Digitalpotentiometer, die dafür eingesetzt werden können.

- Erweiterung der CI

  Die CI kann in der Zukunft dazu genutzt werden, Hardwaretests auf einem Teststand durchzuführen. Außerdem kann die I^2C-Software auf dem Raspberry Pi im Rahmen der Verifikation cross-kompiliert werden.

Literaturverzeichnis

[1] A. P. Davison, E. Müller, *et al.*, *HBP Neuromorphic Computing Platform Guidebook*. Human Brain Project, 2015-2018. EV Guide.

[2] STMicroelectronics, *STM32F405xx Datasheet (DM00037051)*. STMicroelectronics, 2016. STM32F405RG.

[3] STMicroelectronics. st.com [accessed 2018/01/10].

[4] PlatformIO. PlatformIO Docs [accessed 2017/11/14; Revision 448e0f27].

[5] @texane. github: texane/stlink [accessed 2018/03/05].

[6] ARM. PIO - CMSIS.

[7] mbed. PIO - mbed.

[8] STM. PIO - STM32Cube.

[9] Mbed. Mbed OS [accessed 2018/03/05].

[10] waf.io [accessed 2018/03/22].

[11] G. di Sirio. ChibiOS RTOS [accessed 2018/01/11].

[12] G. di Sirio. `http://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:start` [accessed 2018/03/31].

[13] steved. i2c Slave Mode - ChibiOS Forum [accessed 2018/03/27].